Practical Scientific Computing in Python
A Workbook

John D. Hunter
Fernando Pérez

Contents

CHAPTER 1

Introduction

This document contains a set of small problems, drawn from many different fields, meant to illustrate commonly useful techniques for using Python in scientific computing.

All problems are presented in a similar fashion: the task is explained including any necessary mathematical background and a 'code skeleton' is provided that is meant to serve as a starting point for the solution of the exercise. In some cases, some example output of the expected solution, figures or additional hints may be provided as well.

The accompanying source download for this workbook contains the complete solutions, which are not part of this document for the sake of brevity.

For several examples, the provided skeleton contains pre-written tests which validate the correctness of the expected answers. When you have completed the exercise successfully, you should be able to run it from within IPython and see something like this (illustrated using a trapezoidal rule problem, whose solution is in the file `trapezoid.py`):

```
In [7]: run trapezoid.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.003s

OK
```

This message tells you that 4 automatic tests were successfully executed. The idea of including automatic tests in your code is a common one in modern software development, and Python includes in its standard library two modules for automatic testing, with slightly different functionality: `unittest` and `doctest`. These tests were written using the `unittest` system, whose complete documentation can be found here: http://docs.python.org/lib/module-unittest.html.

Other exercises will illustrate the use of the `doctest` system, since it provides complementary functionality.

CHAPTER 2

Simple non-numerical problems

1. Sorting quickly with QuickSort

Quicksort is one of the best known, and probably the simplest, fast algorithm for sorting n items. It is fast in the sense that it requires on average $\mathcal{O}(n \log n)$ comparisons instead of $\mathcal{O}(n^2)$, although a naive implementation does have quadratic worst-case behavior.

The algorithm uses a simple divide and conquer strategy, and its implementation is naturally recursive. Its basic steps are:

(1) Pick an element from the list, called the pivot p (any choice works).
(2) Select from the rest of the list those elements smaller and those greater than the pivot, and store them in separate lists S and G.
(3) Recursively apply the algorithm to S and G. The final result can be written as $\sigma(S) + [p] + \sigma(G)$, where σ represents the sorting operation, $+$ indicates list concatenation and $[p]$ is the list containing the pivot as its single element.

The listing 2.1 contains a skeleton with no implementation but with tests already written (in the form of *unit tests*, as described in the introduction).

LISTING 2.1

```
"""Simple quicksort implementation."""

def qsort(lst):
    """Return a sorted copy of the input list."""

    raise NotImplementedError

if __name__ == '__main__':
    from unittest import main, TestCase
    import random

    class qsortTestCase(TestCase):
        def test_sorted(self):
            seq = range(10)
            sseq = qsort(seq)
            self.assertEqual(seq,sseq)

        def test_random(self):
            tseq = range(10)
            rseq = range(10)
            random.shuffle(rseq)
            sseq = qsort(rseq)
            self.assertEqual(tseq,sseq)
    main()
```

Hints.

- Python has no particular syntactic requirements for implementing recursion, but it does have a maximum recursion depth. This value can be queried

via the function `sys.getrecursionlimit()`, and it can be changed with `sys.setrecursionlimit(new_value)`.
- Like in all recursive problems, don't forget to implement an exit condition!
- If `L` is a list, the call `len(L)` provides its length.

2. Dictionaries for counting words

A common task in text processing is to produce a count of word frequencies. While NumPy has a builtin histogram function for doing numerical histograms, it won't work out of the box for couting discrete items, since it is a binning histogram for a range of real values.

But the Python language provides very powerful string manipulation capabilities, as well as a very flexible and efficiently implemented builtin data type, the *dictionary*, that makes this task a very simple one.

In this problem, you will need to count the frequencies of all the words contained in a compressed text file supplied as input.

The listing 2.2 contains a skeleton for this problem, with `XXX` marking various places that are incomplete.

LISTING 2.2

```
#!/usr/bin/env python
"""Word frequencies - count word frequencies in a string."""

def word_freq(text):
    """Return a dictionary of word frequencies for the given text."""
    # XXX you need to write this

def print_vk(lst):
    """Print a list of value/key pairs nicely formatted in key/value order."""

    # Find the longest key: remember, the list has value/key paris, so the key
    # is element [1], not [0]
    longest_key = max(map(lambda x: len(x[1]),lst))
    # Make a format string out of it
    fmt = '%'+str(longest_key)+'s -> %s'
    # Do actual printing
    for v,k in lst:
        print fmt % (k,v)

def freq_summ(freqs,n=10):
    """Print a simple summary of a word frequencies dictionary.

    Inputs:
      - freqs: a dictionary of word frequencies.

    Optional inputs:
      - n: the number of items to print"""

    words,counts = # XXX look at the keys and values methods of dicts
    # Sort by count

    items = # XXX think of a list, look at zip() and think of sort()

    print 'Number of words:',len(freqs)
    print
    print '%d least frequent words:' % n
```

```
    print_vk(items[:n])
    print
    print '%d most frequent words:' % n
    print_vk(items[-n:])

if __name__ == '__main__':
    text = # XXX
    # You need to read the contents of the file HISTORY.gz.  Do NOT unzip it
    # manually, look at the gzip module from the standard library and  the
    # read() method of file objects.
    freqs = word_freq(text)
    freq_summ(freqs,20)
```

Hints.

- The `print_vk` function is already provided for you as a simple way to summarize your results.
- You will need to read the compressed file `HISTORY.gz`. Python has facilities to do this without having to manually uncompress it.
- Consider 'words' simply the result of splitting the input text into a list, using any form of whitespace as a separator. This is obviously a very naïve definition of 'word', but it shall suffice for the purposes of this exercise.
- Python strings have a `.split()` method that allows for very flexible splitting. You can easily get more details on it in IPython:

```
In [2]: a = 'somestring'

In [3]: a.split?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
Namespace:      Interactive
Docstring:
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator.
```

The complete set of methods of Python strings can be viewed by hitting the TAB key in IPython after typing '`a.`', and each of them can be similarly queried with the '`?`' operator as above. For more details on Python strings and their companion sequence types, see `http://docs.python.org/lib/typesseq.html`.

CHAPTER 3

Working with files, the internet, and numpy arrays

This section is a general overview to show how easy it is to load and manipulate data on the file system and over the web using python's built in data structures and numpy arrays. The goal is to exercise basic programming skills like building filename or web addresses to automate certain tasks like loading a series of data files or downloading a bunch of related files off the web, as well as to illustrate basic numpy and pylab skills.

1. Loading and saving ASCII data

The simplest file format is a plain text ASCII file of numbers. Although there are many better formats out there for saveing and loading data, this format is extremely common because it has the advantages of being human readable, and thus will survive the test of time as the *en vogue* programming languages, analysis applications and data formats come and go, it is easy to parse, and it is supported by almost all languages and applications.

In this exercise we will create a data set of two arrays, the first one regularly sampled time t from 0..2 seconds with 20 ms time step , and the second one an array `v` of sinusoidal voltages corrupted by some noise. Let's assume the sine wave has amplitude 2 V, frequency 10 Hz, and zero mean Gaussian distrubuted white noise with standard deviation 0.5 V. Your task is to write two scripts.

The first script should create the vectors `t` and `v`, plot the time series of `t` versus `v`, save them in a two dimensional numpy array `X`, and then dump the array `X` to a plain text ASCII file called `'noisy_sine.dat'`. The file will look like (not identical because of the noise)

```
0.000000000000000000e+00 1.550947826934816025e-02
2.000000000000000042e-02 2.493944587057004725e+00
4.000000000000000083e-02 9.497694074551737975e-01
5.999999999999999778e-02 -9.185779287524413750e-01
8.000000000000000167e-02 -2.811127590689064704e+00
... and so on
```

Here is the exercise skeleton of the script to create and plot the data file

LISTING 3.1

```
from scipy import arange, sin, pi, randn, zeros
import pylab as p

a = 2       # 2 volt amplitude
f = 10      # 10 Hz frequency
sigma = 0.5 # 0.5 volt standard deviation noise

# create the t and v arrays; see the scipy commands arange, sin, and randn
t = XXX # an evenly sampled time array
v = XXX # a noisy sine wave

# create a 2D array X and put t in the 1st column and v in the 2nd;
# see the numpy command zeros
X = XXX
```

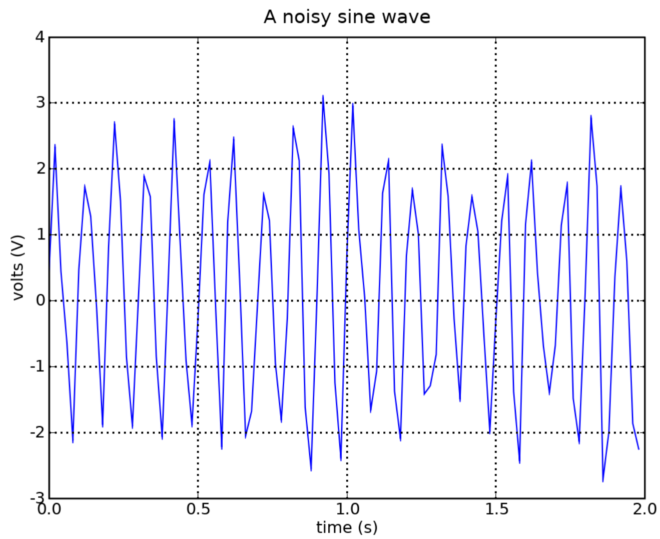

FIGURE 1. A 10 Hz sine wave corrupted by noise

```
# save the output file as ASCII; see the pylab command save
XXX

# plot the arrays t vs v and label the x-axis, y-axis and title save
# the output figure as noisy_sine.png.  See the pylab commands plot,
# xlabel, ylabel, grid, show
XXX
```

and the graph will look something like Figure 1

The second part of this exercise is to write a script which loads data from the data file into an array `X`, extracts the columns into arrays `t` and `v`, and computes the RMS (root-mean-square) intensity of the signal using the `load` command.

2. Working with CSV files

The CSV (Comma Separated Value) file specification is also an ASCII based, human readable format, but it is more powerful than simple flat ASCII files including headers, escape sequences and arbitrary delimiters like TAB, SPACE or COMMA. It is a widely used interchange format for sharing data between operating systems and programs like Excel, Matlab and statistical analysis packages.

A typical CSV file will be a mix of different data types: integers, floating point numbers, dates and strings. Of course, all of these are strings in the file, since all text files are made up of strings, but the data is typically representing some other numeric or date type. Python has very good support for handling different data types, so you don't need to try to force your data to look like a multi dimensional array of floating point numbers if this is not the natural way to describe your data. numpy provides a generalization of the array data structure we used above called record arrays, which allow to store data in a conceptual model similar to a database or spreadsheet: several named fields (eg 'date', 'weight', 'height', 'age') with different types (eg `datetime.date`, `float`, `float`, `int`).

In the example below, we will download some CSV files from Yahoo Financial web pages and load them into numpy record arrays for analysis and visualization. Go to `http://finance.yahoo.com` and enter a stock symbol in the entry boc labeled "Get Quotes". I will use `'SPY'` which is an index fund that tracks the S&P 500. In the left menu bar, there is an entry called "Historical Prices" which will take you to a page where you can download the price history of your stock. Near the bottom of this page you should see a "Download To Spreadsheet" link – instead of clicking on it, right click it and choose "Copy Link Location" and paste this into a python script or ipython session as a string named `url`. Eg, for SPY page better

```
url = 'http://ichart.finance.yahoo.com/table.csv?' +\
   's=SPY&d=9&e=20&f=2007&g=d&a=0&b=29&c=1993&ignore=.csv'
```

I've broken the url into two strings so they will fit on the page. If you spend a little time looking at this pattern, you can probably figure out what is going on. The URL is encoding the information about the stock, the variable `s` for the stock ticker, `d` for the latest month, `e` for the latest day, `f` for the latest year, `c` for the start year, and so on (similarly `a`, `b`, and `c` for the start month, day and year). This is handy to know, because below we will write some code to automate some downloads for a stock universe.

One of the great things about python is it's "batteries included" standard library, which includes support for dates, csv files and internet downloads. The example interactive session below shows how in just a few lines of code using python's `urllib` for retrieving information from the internet, and matplotlib's `csv2rec` function for loading numpy record arrays, we are ready to get to work analyzing some web based data. Comments have been added to a copy-and-paste from the interactive session

```
# import a couple of libraries we'll be needing
In [23]: import urllib
In [24]: import matplotlib.mlab as mlab

# this is the CSV file we'll be downloading
In [25]: url = 'http://ichart.finance.yahoo.com/table.csv?' +\
   's=SPY&d=9&e=20&f=2007&g=d&a=0&b=29&c=1993&ignore=.csv'

# this will grab that web file and save it as 'SPY.csv' on our local
# filesystem
In [27]: urllib.urlretrieve(url, 'SPY.csv')
Out[27]: ('SPY.csv', <httplib.HTTPMessage instance at 0x2118210>)

# here we use the UNIX command head to peak into the file, which is
# a comma separated and contains various types, dates, ints, floats
In [28]: !head SPY.csv
Date,Open,High,Low,Close,Volume,Adj Close
2007-10-19,153.09,156.48,149.66,149.67,295362200,149.67
2007-10-18,153.45,154.19,153.08,153.69,148367500,153.69
2007-10-17,154.98,155.09,152.47,154.25,216687300,154.25
2007-10-16,154.41,154.52,153.47,153.78,166525700,153.78
2007-10-15,156.27,156.36,153.94,155.01,161151900,155.01
2007-10-12,155.46,156.35,155.27,156.33,124546700,156.33
2007-10-11,156.93,157.52,154.54,155.47,233529100,155.47
2007-10-10,156.04,156.44,155.41,156.22,101711100,156.22
2007-10-09,155.60,156.50,155.03,156.48,94054300,156.48

# csv2rec will import the file into a numpy record array, inspecting
# the columns to determine the correct data type
In [29]: r = mlab.csv2rec('SPY.csv')

# the dtype attribute shows you the field names and data types.
# O4 is a 4 byte python object (datetime.date), f8 is an 8 byte
# float, i4 is a 4 byte integer and so on.  The > and < symbols
# indicate the byte order of multi-byte data types, eg big endian or
```

```
# little endian, which is important for cross platform binary data
# storage
In [30]: r.dtype
Out[30]: dtype([('date', '|O4'), ('open', '>f8'), ('high', '>f8'),
('low', '>f8'), ('close', '>f8'), ('volume', '>i4'), ('adj_close',
'>f8')])

# Each of the columns is stored as a numpy array, but the types are
# preserved.  Eg, the adjusted closing price column adj_close is a
# floating  point type, and the date column is a python datetime.date
In [31]: print r.adj_close
[ 149.67  153.69  154.25 ...,    34.68   34.61   34.36]
In [32]: print r.date
[2007-10-19 00:00:00 2007-10-18 00:00:00 2007-10-17 00:00:00 ...,
 1993-02-02 00:00:00 1993-02-01 00:00:00 1993-01-29 00:00:00]
```

For your exercise, you'll elaborate on the code here to do a batch download of a number of stock tickers in a defined stock universe. Define a function `fetch_stock(ticker)` which takes a stock ticker symbol as an argument and returns a numpy record array. Select the rows of the record array where the date is greater than 2003-01-01 and plot the returns $(p - p_0)/p_0$ where p are the prices and p_0 is the initial price. by date for each stock on the same plot. Create a legend for the plot using the matplotlib `legend` command, and print out a sorted list of final returns (eg assuming you bought in 2003 and held to the present) for each stock. Here is the exercise skeleton.:

Listing 3.2

```
"""
Download historical pricing record arrays for a universe of stocks
from Yahoo Finance using urllib.  Load them into numpy record arrays
using matplotlib.mlab.csv2rec, and do some batch processing -- make
date vs price charts for each one, and compute the return since 2003
for each stock.  Sort the returns and print out the tickers of the 4
biggest winners
"""
import os, datetime, urllib
import matplotlib.mlab as mlab  # contains csv2rec
import numpy as npy
import pylab as p

def fetch_stock(ticker):
    """
    download the CSV file for stock with ticker and return a numpy
    record array.  Save the CSV file as TICKER.csv where TICKER is the
    stock's ticker symbol.

    Extra credit for supporting a start date and end date, and
    checking to see if the file already exists on the local file
    system before re-downloading it
    """
    fname = '%s.csv'%ticker
    url = XXX # create the url for this ticker

    # the os.path module contains function for checking whether a file
    # exists, and fetch it if not
    XXX
```

```
    # load the CSV file intoo a numpy record array
    r = XXX

    # note that the CSV file is sorted most recent date first, so you
    # will probably want to sort the record array so most recent date
    # is last
    XXX
    return r

tickers = 'INTC', 'MSFT', 'YHOO', 'GOOG', 'GE', 'WMT', 'AAPL'

# we want to compute returns since 2003, so define the start date as a
# datetime.datetime instance
startdate = XXX

# we'll store a list of each return and ticker for analysis later
data = []   # a list of (return, ticker) for each stock
fig = p.figure()
for ticker in tickers:
    print 'fetching', ticker
    r = fetch_stock(ticker)

    # select the numpy records where r.date>=startdatre use numpy mask
    # indexing to restrict r to just the dates > startdate
    r = XXX
    price = XXX   # set price equal to the adjusted close
    returns = XXX # return is the (price-p0)/p0
    XXX           # store the data

    # plot the returns by date for each stock using pylab.plot, adding
    # a label for the legend
    XXX

# use pylab legend command to build a legend
XXX

# now sort the data by returns and print the results for each stock
XXX

# show the figures
p.show()
```

The graph will look something like Figure 2.

3. Loading and saving binary data

ASCII is bloated and slow for working with large arrays, and so binary data should be used if performance is a consideration. To save an array X in binary form, you can use the numpy `tostring` method

```
In [16]: import numpy

# create some random numbers
In [17]: x = numpy.random.rand(5,2)
```

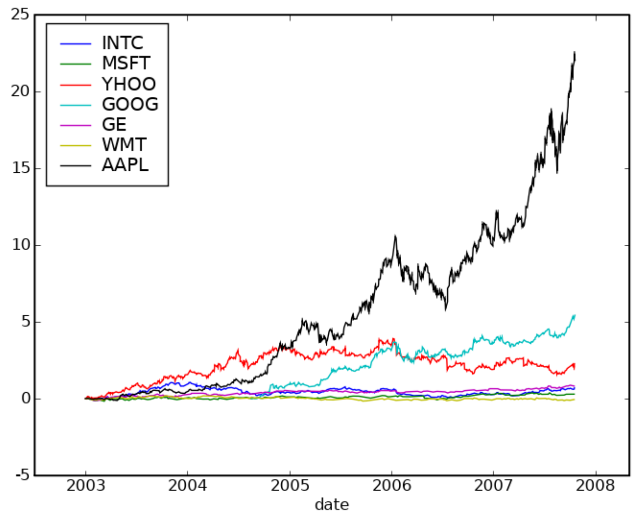

FIGURE 2. Returns for a universe of stocks since 2003

```
In [19]: print x
[[ 0.56331918  0.519582  ]
 [ 0.22685429  0.18371135]
 [ 0.19384767  0.27367054]
 [ 0.35935445  0.95795884]
 [ 0.37646642  0.14431089]]

# save it to a data file in binary
In [20]: x.tofile(file('myx.dat', 'wb'))

# load it into a new array
In [21]: y = numpy.fromfile(file('myx.dat', 'rb'))

# the shape is not preserved, so we will have to reshape it
In [22]: print y
[ 0.56331918  0.519582    0.22685429  0.18371135  0.19384767
0.27367054
  0.35935445  0.95795884  0.37646642  0.14431089]

In [23]: y.shape
Out[23]: (10,)

# restore the original shape
In [24]: y.shape = 5, 2

In [25]: print y
[[ 0.56331918  0.519582  ]
 [ 0.22685429  0.18371135]
 [ 0.19384767  0.27367054]
 [ 0.35935445  0.95795884]
 [ 0.37646642  0.14431089]]
```

The advantage of numpy `tofile` and `fromfile` over ASCII data is that the data storage is compact and the read and write are very fast. It is a bit of a pain that that meta ata like array datatype and shape are not stored. In this format, just the raw binary numeric data is stored, so you will have to keep track of the data type and shape by other means. This is a good solution if you need to port binary data files between different packages, but if you know you will always be working in python, you can use the python pickle function to preserve all metadata (pickle also works with all standard python data types, but has the disadvantage that other programs and applications cannot easily read it)

```
# create a 6,3 array of random integers
In [36]: x = (256*numpy.random.rand(6,3)).astype(numpy.int)

In [37]: print x
[[173  38   2]
 [243 207 155]
 [127  62 140]
 [ 46  29  98]
 [  0  46 156]
 [ 20 177  36]]

# use pickle to save the data to a file myint.dat
In [38]: import cPickle

In [39]: cPickle.dump(x, file('myint.dat', 'wb'))

# load the data into a new array
In [40]: y = cPickle.load(file('myint.dat', 'rb'))

# the array type and share are preserved
In [41]: print y
[[173  38   2]
 [243 207 155]
 [127  62 140]
 [ 46  29  98]
 [  0  46 156]
 [ 20 177  36]]
```

CHAPTER 4

Elementary Numerics

1. Wallis' slow road to π

Wallis' formula is an infinite product that converges (slowly) to π:

$$\pi = \prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}. \tag{1}$$

The listing 4.1 contains a skeleton with no implementation but with some plotting commands already inserted, so that you can visualize the convergence rate of this formula as more terms are kept.

LISTING 4.1

```
#!/usr/bin/env python
"""Simple demonstration of Python's arbitrary-precision integers."""

# We need exact division between integers as the default, without manual
# conversion to float b/c we'll be dividing numbers too big to be represented
# in floating point.
from __future__ import division

from decimal import Decimal

def pi(n):
    """Compute pi using n terms of Wallis' product.

    Wallis' formula approximates pi as

    pi(n) = 2 \prod_{i=1}^{n}\frac{4i^2}{4i^2-1}."""

    XXX

# This part only executes when the code is run as a script, not when it is
# imported as a library
if __name__ == '__main__':
    # Simple convergence demo.

    # A few modules we need
    import pylab as P
    import numpy as N

    # Create a list of points 'nrange' where we'll compute Wallis' formula
    nrange =  XXX

    # Make an array of such values
    wpi =  XXX
    # Compute the difference against the value of pi in numpy (standard
```

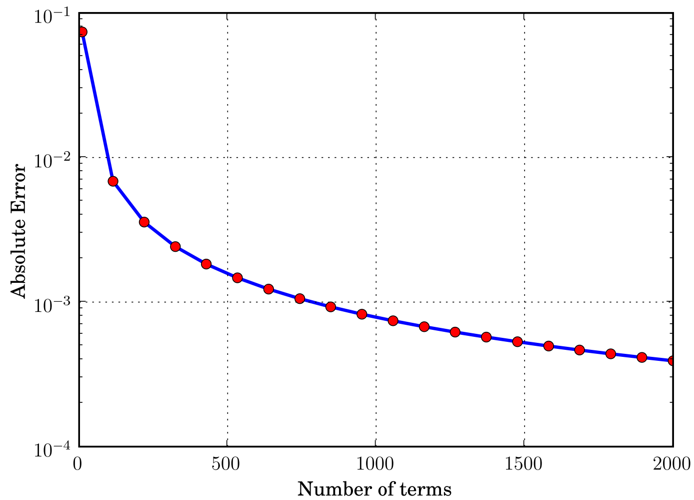

FIGURE 1. Convergence rate for Wallis' infinite product approximation to π.

```
# 16-digit value)
diff =  XXX

# Make a new figure and build a semilog plot of the difference so we can
# see the quality of the convergence

P.figure()
# Line plot with red circles at the data points
P.semilogy(nrange,diff,'-o',mfc='red')

# A bit of labeling and a grid
P.title(r"Convergence of Wallis' product formula for pi")
P.xlabel('Number of terms')
P.ylabel(r'|Error}|')
P.grid()

# Display the actual plot
P.show()
```

After running the script successfully, you should obtain a plot similar to Figure 1.

2. Trapezoidal rule

In this exercise, you are tasked with implementing the simple trapezoid rule formula for numerical integration. If we want to compute the definite integral

$$\int_a^b f(x)dx \tag{2}$$

we can partition the integration interval $[a, b]$ into smaller subintervals, and approximate the area under the curve for each subinterval by the area of the trapezoid created by linearly interpolating between the two function values at each end of the subinterval. This is graphically illustrated in Figure 2, where the blue line represents the function $f(x)$ and the red line represents the successive linear segments.

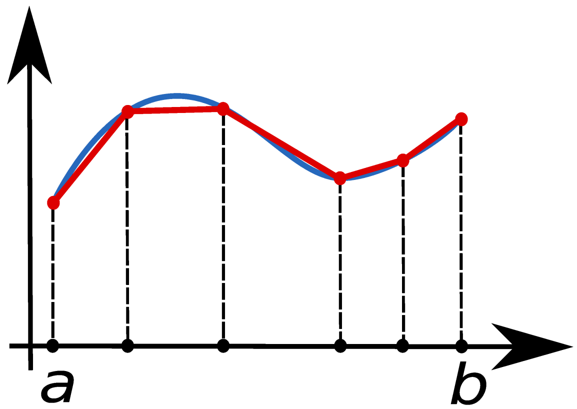

FIGURE 2. Illustration of the composite trapezoidal rule with a non-uniform grid (Image credit: Wikipedia).

The area under $f(x)$ (the value of the definite integral) can thus be approximated as the sum of the areas of all these trapezoids. If we denote by x_i ($i = 0, \dots, n$, with $x_0 = a$ and $x_n = b$) the abscissas where the function is sampled, then

$$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{i=1}^{n}(x_i - x_{i-1})\left(f(x_i) + f(x_{i+1})\right). \tag{3}$$

The common case of using equally spaced abscissas with spacing $h = (b-a)/n$ reads simply

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=1}^{n}\left(f(x_i) + f(x_{i+1})\right). \tag{4}$$

One frequently receives the function values already precomputed, $y_i = f(x_i)$, so equation (3) becomes

$$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{i=1}^{n}(x_i - x_{i-1})\left(y_i + y_{i-1}\right). \tag{5}$$

Listing 4.2 contains a skeleton for this problem, written in the form of two incomplete functions and a set of automatic tests (in the form of *unit tests*, as described in the introduction).

LISTING 4.2

```python
#!/usr/bin/env python
"""Simple trapezoid-rule integrator."""

import numpy as N

def trapz(x, y):
    """Simple trapezoid integrator for sequence-based innput.

    Inputs:
      - x,y: arrays of the same length.

    Output:
      - The result of applying the trapezoid rule to the input, assuming that
      y[i] = f(x[i]) for some function f to be integrated.

    Minimally modified from matplotlib.mlab."""
    raise NotImplementedError
```

```python
def trapzf(f,a,b,npts=100):
    """Simple trapezoid-based integrator.

    Inputs:
      - f: function to be integrated.

      - a,b: limits of integration.

    Optional inputs:
      - npts(100): the number of equally spaced points to sample f at, between
      a and b.

    Output:
      - The value of the trapezoid-rule approximation to the integral."""
    raise NotImplementedError

if __name__ == '__main__':
    # Simple tests for trapezoid integrator, when this module is called as a
    # script from the command line.

    import unittest
    import numpy.testing as ntest

    def square(x): return x**2

    class trapzTestCase(unittest.TestCase):
        def test_err(self):
            self.assertRaises(ValueError,trapz,range(2),range(3))

        def test_call(self):
            x = N.linspace(0,1,100)
            y = N.array(map(square,x))
            ntest.assert_almost_equal(trapz(x,y),1./3,4)

    class trapzfTestCase(unittest.TestCase):
        def test_square(self):
            ntest.assert_almost_equal(trapzf(square,0,1),1./3,4)

        def test_square2(self):
            ntest.assert_almost_equal(trapzf(square,0,3,350),9.0,4)

    unittest.main()
```

In this exercise, you'll need to write two functions, `trapz` and `trapzf`. `trapz` applies the trapezoid formula to pre-computed values, implementing equation (5), while `trapzf` takes a function f as input, as well as the total number of samples to evaluate, and computes eq. (4).

3. Newton's method

Consider the problem of solving for t in

$$\int_o^t f(s)ds = u \tag{6}$$

where $f(s)$ is a monotonically increasing function of s and $u > 0$.

This problem can be simply solved if seen as a root finding question. Let

$$g(t) = \int_o^t f(s)ds - u, \tag{7}$$

then we just need to find the root for $g(t)$, which is guaranteed to be unique given the conditions above.

The SciPy library includes an optimization package that contains a Newton-Raphson solver called `scipy.optimize.newton`. This solver can optionally take a known derivative for the function whose roots are being sought, and in this case the derivative is simply

$$\frac{dg(t)}{dt} = f(t). \tag{8}$$

For this exercise, implement the solution for the test function

$$f(t) = t\sin^2(t),$$

using

$$u = \frac{1}{4}.$$

The listing 4.3 contains a skeleton that includes for comparison the correct numerical value.

LISTING 4.3

```python
#!/usr/bin/env python
"""Root finding using SciPy's Newton's method routines.
"""

from math import sin

import scipy, scipy.integrate, scipy.optimize

quad = scipy.integrate.quad
newton = scipy.optimize.newton

# test input function f(t): t * sin^2(t)
def f(t): XXX

# Use u=0.25
def g(t): XXX

# main
tguess = 10.0

print "Solution using the numerical integration technique"
t0 = newton(g,tguess,f)
print "t0, g(t0) =",t0,g(t0)

print
print "To six digits, the answer in this case is t==1.06601."
```

CHAPTER 5

Linear algebra

Like matlab, numpy and scipy have support for fast linear algebra built upon the highly optimized LAPACK, BLAS and ATLAS fortran linear algebra libraries. Unlike Matlab, in which everything is a matrix or vector, and the '*' operator always means matrix multiple, the default object in numpy is an `array`, and the '*' operator on arrays means element-wise multiplication.

Instead, numpy provides a `matrix` class if you want to do standard matrix-matrix multiplication with the '*' operator, or the `dot` function if you want to do matrix multiplies with plain arrays. The basic linear algebra functionality is found in `numpy.linalg`

```
In [1]: import numpy as npy
In [2]: import numpy.linalg as linalg

# X and Y are arrays
In [3]: X = npy.random.rand(3,3)
In [4]: Y = npy.random.rand(3,3)

# * operator is element wise multiplication, not matrix matrix
In [5]: print X*Y
[[ 0.00973215  0.18086148  0.05539387]
 [ 0.00817516  0.63354021  0.2017993 ]
 [ 0.34287698  0.25788149  0.15508982]]

# the dot function will use optimized LAPACK to do matrix-matix
# multiply
In [6]: print npy.dot(X, Y)
[[ 0.10670678  0.68340331  0.39236388]
 [ 0.27840642  1.14561885  0.62192324]
 [ 0.48192134  1.32314856  0.51188578]]

# the matrix class will create matrix objects that support matrix
# multiplication with *
In [7]: Xm = npy.matrix(X)
In [8]: Ym = npy.matrix(Y)
In [9]: print Xm*Ym
[[ 0.10670678  0.68340331  0.39236388]
 [ 0.27840642  1.14561885  0.62192324]
 [ 0.48192134  1.32314856  0.51188578]]

# the linalg module provides functions to compute eigenvalues,
# determinants, etc.  See help(linalg) for more info
In [10]: print linalg.eigvals(X)
[ 1.46131600+0.j          0.46329211+0.16501143j  0.46329211-0.16501143j]
```

1. Glass Moiré Patterns

When a random dot pattern is scaled, rotated, and superimposed over the original dots, interesting visual patterns known as Glass Patterns emerge[1] In this exercise, we generate random dot fields using numpy's uniform distribution function, and apply transformations to the random dot field using a scale $\mathbf{S}$ and rotation $\mathbf{R}$ matrix $\mathbf{X_2} = \mathbf{SRX_1}$.

If the scale and rotation factors are small, the transformation is analogous to a single step in the numerical solution of a 2D ODE, and the plot of both $\mathbf{X_1}$ and $\mathbf{X_2}$ will reveal the structure of the vecotr field flow around the fixed point (the invariant under the transformation); see for example the *stable focus*, aka *spiral*, in Figure 1.

The eigenvalues of the tranformation matrix $\mathbf{M} = \mathbf{SR}$ determine the type of fix point: *center*, *stable focus*, *saddle node*, etc.... For example, if the two eigenvalues are real but differing in signs, the fixed point is a *saddle node*. If the real parts of both eigenvalues are negative and the eigenvalues are complex, the fixed point is a *stable focus*. The complex part of the eigenvalue determines whether there is any rotation in the matrix transformation, so another way to look at this is to break out the scaling and rotation components of the transformation $\mathbf{M}$. If there is a rotation component, then the fixed point will be a *center* or a *focus*. If the scaling components are both one, the rotation will be a *center*, if they are both less than one (contraction), it will be a *stable focus*. Likewise, if there is no rotation component, the fixed point will be a *node*, and the scaling components will determine the type of node. If both are less than one, we have a *stable node*, if one is greater than one and the other less than one, we have a *saddle node*.

LISTING 5.1

```
"""
Moire patterns from random dot fields

http://en.wikipedia.org/wiki/Moir%C3%A9_pattern

See L. Glass. 'Moire effect from random dots' Nature 223, 578580 (1969).
"""
from numpy import cos, sin, pi, matrix
import numpy as npy
import numpy.linalg as linalg
from pylab import figure, show

def csqrt(x):
    'sqrt func that handles returns sqrt(x)j for x<0'
    XXX

def myeig(M):
    """
    compute eigen values and eigenvectors analytically
    Solve quadratic:
      lamba^2 - tau*lambda + Delta = 0
    where tau = trace(M) and Delta = Determinant(M)

    Return value is lambda1, lambda2
    """
    XXX

# 2000 random x,y points in the interval[-0.5 ... 0.5]
X1 = XXX
```

[1] L. Glass. 'Moiré effect from random dots' Nature 223, 578580 (1969).

```
name =  'saddle'
#sx, sy, angle = XXX

#name = 'center'
#sx, sy, angle = XXX

name = 'spiral'  #stable focus
sx, sy, angle = XXX

theta = angle * pi/180.  # the rotation in radians


# the scaling matrix
# | sx 0 |
# | 0 sy |
S = XXX

# the rotation matrix
# | cos(theta)  -sin(theta) |
# | sin(theta)  cos(theta)  |
R = XXX

# the transformation is the matrix product of the scaling and rotation
M = XXX

# compute the eigenvalues using numpy linear algebra
print 'numpy eigenvalues', XXX

# compare with the analytic values from myeig
print 'analytic eigenvalues', myeig(M)

# transform X1 by the matrix M
X2 = XXX

# plot the original X1 as green dots and the transformed X2 as red
# dots
XXX
```

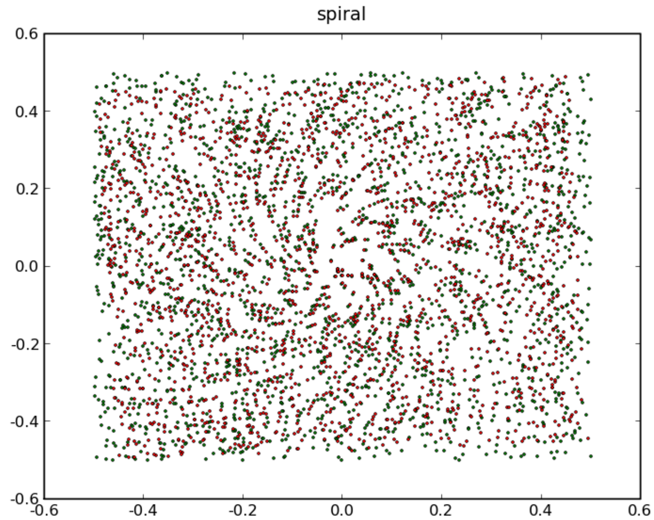

FIGURE 1. Glass pattern showing a stable focus

CHAPTER 6

Signal processing

numpy and scipy provide many of the essential tools for digital signal processing. scipy.signal provides basic tools for digital filter design and filtering (eg Butterworth filters), a linear systems toolkit, standard waveforms such as square waves, and saw tooth functions, and some basic wavelet functionality. scipy.fftpack provides a suite of tools for Fourier domain analysis, including 1D, 2D, and ND discrete fourier transform and inverse functions, in addition to other tools such as analytic signal representations via the Hilbert trasformation (numpy.fft also provides basic FFT functions). pylab provides Matlab compatible functions for computing and plotting standard time series analyses, such as historgrams (hist), auto and cross correlations (acorr and xcorr), power spectra and coherence spectra (psd, csd, cohere and specgram.

LISTING 6.1

```
"""
In signal processing, the output of a linear system to an arbitrary
input is given by the convolution of the impule response function (the
system response to a Dirac-delta impulse) and the input signal.

Mathematically:

  y(t) = \int_0^\t x(\tau)r(t-\tau)d\tau
```

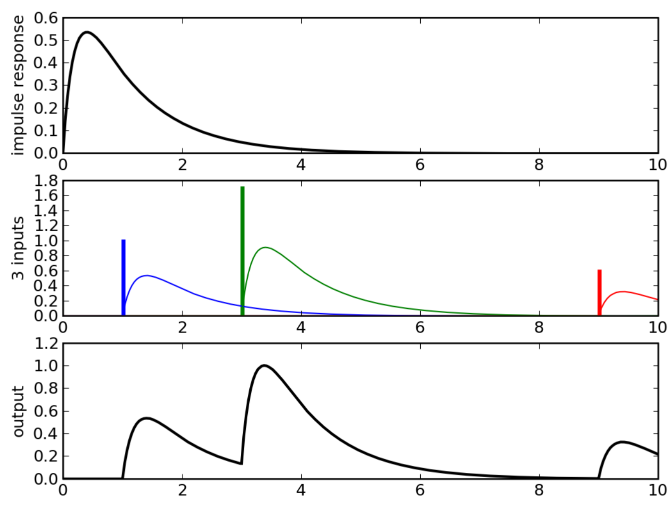

FIGURE 1. The output of a linear system to a series of impulse inputs is equal to the sum of the scaled and time shifted impulse response functions.

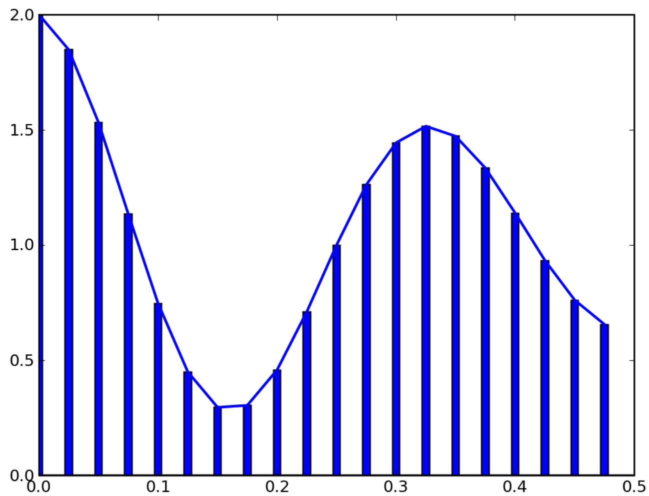

FIGURE 2. Representing a continuous time signal sampled as a sum of delta functions.

```
where x(t) is the input signal at time t, y(t) is the output, and r(t)
is the impulse response function.

In this exercise, we will compute investigate the convolution of a
white noise process with a double exponential impulse response
function, and compute the results

  * using numpy.convolve

  * in Fourier space using the property that a convolution in the
    temporal domain is a multiplication in the fourier domain
"""

import numpy as npy
import matplotlib.mlab as mlab
from pylab import figure, show

# build the time, input, output and response arrays
dt = 0.01
t = XXX         # the time vector from 0..20
Nt = len(t)

def impulse_response(t):
    'double exponential response function'
    return XXX


x = XXX   # gaussian white noise

# evaluate the impulse response function, and numerically convolve it
# with the input x
```

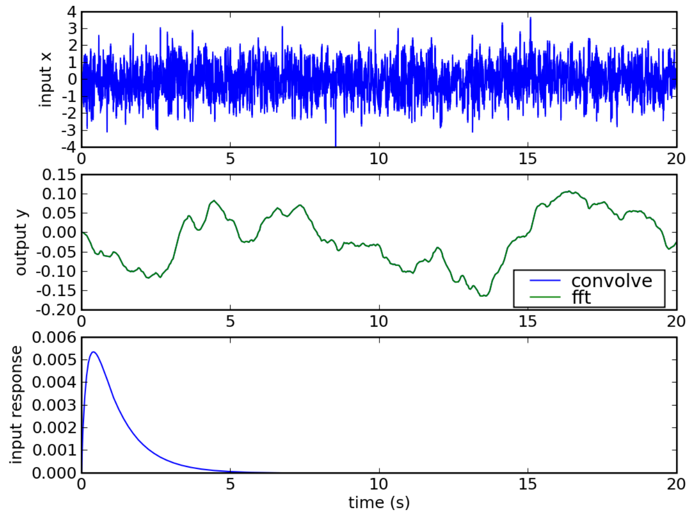

FIGURE 3. Convolution of a white noise process with a double exponential function computed with numpy.fft and numpy.convolve

```
r = XXX # evaluate the impulse function
y = XXX # convultion of x with r
y = XXX # extract just the length Nt part

# compute y by applying F^-1[F(x) * F(r)].  The fft assumes the signal
# is periodic, so to avoid edge artificats, pad the fft with zeros up
# to the length of r + x do avoid circular convolution artifacts
R = XXX  # the zero padded FFT of r
X = XXX  # the zero padded FFT of x
Y = XXX  # the product of R and S

# now inverse fft and extract the real part, just the part up to
# len(x)
yi = XXX

# plot t vs x, t vs y and yi, and t vs r in three subplots
XXX
show()
```

Convolution of an input with with a linear filter in the termporal or spatial domain is equivalent to multiplication by the fourier transforms of the input and the filter in the spectral domain. This provides a conceptually simple way to think about filtering: transform your signal into the frequency domain, dampen the frequencies you are not interested in by multiplying the frequency spectrum by the desired weights, and then inverse transform the multiplies spectrum back into the original domain. In the example below, we will simply set the weights of the frequencies we are uninterested in (the high frequency noise) to zero rather than dampening them with a smoothly varying function. Although this is not usually the best thing to do, since sharp edges in one domain usually introduce artifacts in another (eg high frequency "ringing"), it is easy to do and sometimes provides satisfactory results.

The image in the upper left panel of Figure 4 is a grayscale photo of the moon landing. There is a banded pattern of high frequency noise polluting the image. In the upper right panel we see the 2D spatial frequency spectrum. The FFT output in `scipy` is packed with the lower freqeuencies starting in the upper left, and proceeding to higher frequencies as one moves to the center of the spectrum (this is the most efficient way numerically to fill the output of the FFT algorithm). Because the input signal is real, the output spectrum is complex and symmetrical: the transformation values beyond the midpoint of the frequency spectrum (the Nyquist frequency) correspond to the values for negative frequencies and are simply the mirror image of the positive frequencies below the Nyquist (this is true for the 1D, 2D and ND FFTs in `numpy`).

In this exercise we will compute the 2D spatial frequency spectra of the luminance image, zero out the high frequency components, and inverse transform back into the time domain. We can plot the input and output images with the `pylab.imshow` function, but the images must first be scaled to be withing the 0..1 luminance range. For best results, it helps to *amplify* the image by some scale factor, and then *clip* it to set all values greater than one to one. This serves to enhance contrast among the darker elements of the image, so it is not completely dominated by the brighter segments

Listing 6.2

```python
#!/usr/bin/env python
"""Image denoising example using 2-dimensional FFT."""

import numpy as N
import pylab as P
import scipy as S

def mag_phase(F):
    """Return magnitude and phase components of spectrum F."""

    # XXX Look at the absolute and angle functions in numpy...

def plot_spectrum(F, amplify=1000):
    """Normalise, amplify and plot an amplitude spectrum."""

    M = # XXX use mag_phase to get the magnitude...

    # XXX Now, rescale M by amplify/maximum_of_M.  Numpy arrays can be scaled
    # in-place with ARR *= number.  For the max of an array, look for its max
    # method.

    # XXX Next, clip all values larger than one to one.  You can set all
    # elements of an array which satisfy a given condition with array indexing
    # syntax: ARR[ARR<VALUE] = NEWVALUE, for example.


    # Display: this one already works, if you did everything right with M
    P.imshow(M, P.cm.Blues)


# 'main' script


im = # XXX make an image array from the file 'moonlanding.png', using the
     # pylab imread() function.  You will need to just extract the red
     # channel from the MxNx4 RGBA matrix to represent the grayscale
     # intensities
```

```
F = # Compute the 2d FFT of the input image.  Look for a 2-d FFT in N.dft

# Define the fraction of coefficients (in each direction) we keep
keep_fraction = 0.1

# XXX Call ff a copy of the original transform.  Numpy arrays have a copy
    ...method
# for this purpose.

# XXX Set r and c to be the number of rows and columns of the array.  Look for
# the shape attribute...

# Set to zero all rows with indices between r*keep_fraction and
# r*(1-keep_fraction):

# Similarly with the columns:


# Reconstruct the denoised image from the filtered spectrum.  There's an
# inverse 2d fft in the dft module as well. Call the result im_new

# Show the results.

# The code below already works, if you did everything above right.
P.figure()

P.subplot(221)
P.title('Original image')
P.imshow(im, P.cm.gray)

P.subplot(222)
P.title('Fourier transform')
plot_spectrum(F)

P.subplot(224)
P.title('Filtered Spectrum')
plot_spectrum(ff)

P.subplot(223)
P.title('Reconstructed Image')
P.imshow(im_new, P.cm.gray)

P.show()
```

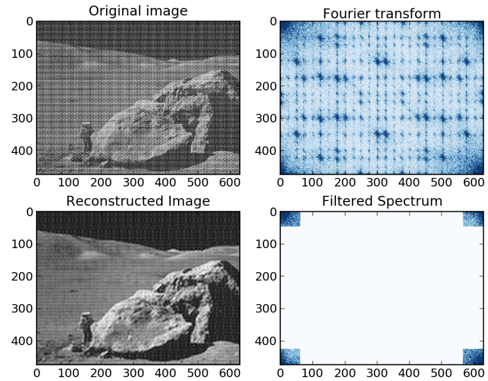

FIGURE 4. High freqeuency noise filtering of a 2D image in the Fourier domain. The upper panels show the original image (left) and spectral power (right) and the lower panels show the same data with the high frequency power set to zero. Although the input and output images are grayscale, you can provide colormaps to `pylab.imshow` to plot them in psudo-color